МИНИСТЕРСТВО НАУКИ И ВЫСШЕГО ОБРАЗОВАНИЯ РОССИЙСКОЙ ФЕДЕРАЦИИ
Озерский технологический институт — филиал НИЯУ МИФИ

Кафедра прикладной математики

Вл. Пономарев

Практикум 2

по теории языков программирования и методам трансляции

Учебно-методическое пособие

Трансляция языков программирования

Озерск, 2019

УДК 681.3.06
П 56

Вл. Пономарев. Практикум 2 по теории языков программирования и методам трансляции. Учебно-методическое пособие. Трансляция языков программирования. Озерск: ОТИ НИЯУ МИФИ, 2019. — 53 с.

В пособии подробно излагается, как выполнять практические работы по дисциплине «Теория языков программирования и методы трансляции». Работы второго семестра изучения дисциплины включают в себя лексический анализ и синтаксический разбор языков программирования.

В качестве основного материала при выполнении практических работ пособие предназначено для студентов, обучающихся по направлению подготовки 09.03.01 «Информатика и вычислительная техника» и специальности 09.05.01 «Применение и эксплуатация автоматизированных систем специального назначения».

Рецензенты:
1.
2.

УТВЕРЖДЕНО
Редакционно-издательским
Советом ОТИ НИЯУ МИФИ

Содержание

Общие цели занятий

В ходе практических работ данной части изучаются лексический анализ и синтаксический разбор. Темы работ включают в себя:

- разработку конечных автоматов,
- формирование потока токенов,
- разработку синтаксических грамматик,
- нисходящий анализ, в том числе методом LL(1),
- восходящий анализ методом LR(1),
- трансляционные грамматики,
- атрибутный перевод.

Программирование ведется в среде Microsoft Visual C++ на языке программирования C++. Рабочее пространство для проведения работ представляет собой консольное приложение simplet (simple translator).

Для защиты знаний и навыков, полученных в ходе выполнения практических работ студент должен иметь тетрадь (12-18 листов). Для каждой выполненной работы в тетрадь записывается отчет.

Отчет по работе начинается с заголовка:

1. Фамилия Имя Отчество
2. Группа
3. Дата начала выполнения работы
4. Код работы
5. Название работы
6. Цели работы
7. Задачи работы

При необходимости при выполнении работы в отчет записываются контрольные значения. Во время защиты тетрадь используется для вопросов преподавателя и ответов студента.

1. Работа ST-101. Лексический анализ

Цели:
- разработка конечных автоматов.

Задачи:
- конечный автомат для разбора комментария Си;
- конечный автомат для приема идентификатора;
- конечные автоматы для разбора числа;
- конечный автомат для разбора строкового литерала.

Опорные документы:
[1]

1.1. Схема лексического анализа

Лексический анализ включает в себя обнаружение и выделение лексем, обработку лексических ошибок, формирование потока токенов.

Под обнаружением лексемы будем подразумевать определение типа лексемы, то есть ее левой границы. Под выделением лексемы будем подразумевать определение текста лексемы, то есть ее правой границы.

В целях упрощения будем рассматривать только такие лексемы, левая граница которых определяется первым символом текста лексемы. Кроме того, обработку ошибочных лексем мы не рассматриваем, и строим непрямой лексический анализатор. Тогда лексический анализ в целом можно представить следующим образом. Некая процедура анализирует входной поток и определяет первый значащий символ. Назовем эту процедуру циклом предварительного анализа. Как только значащий символ будет обнаружен, вызывается функция конечного автомата, распознающего лексему, соответствующую значащему символу. Функция разбирает лексему, и управление возвращается к поиску следующего значащего символа.

Значащими являются символы, которые могут быть первыми символами текста лексем. Незначащими являются пробельные символы, которые просто пропускаются, а также символ перевода строки, код 10. Пробельные символы, — это собственно пробел, код 32, табулятор, код 9, возврат каретки, код 13. Перевод строки в одних случаях пропускается, в других формирует токен, но в любом случае он выделяется для того, чтобы подсчитывать номер строки входного текста. Кроме того, символ конца файла EOF, код –1, ведет к формированию специального токена, обозначающего конец потока.

Такой лексический анализатор называется непрямым, так как он состоит из множества конечных автоматов, а не из одного автомата, который разбирает все возможные лексемы одновременно.

Кроме того, различают последовательное и параллельное взаимодействие лексического и синтаксического анализаторов.

При последовательном взаимодействии входной текст обрабатывается лексическим анализатором от начала до конца, и формируется поток токенов в виде массива, связного списка, или, лучше, файла. Синтаксический анализатор запускается вторым, и по мере необходимости запрашивает очередной токен при помощи функции next_token.

При параллельном взаимодействии работу начинает синтаксический анализатор. По мере необходимости он запрашивает у лексического анализатора очередной токен при помощи функции той же функции next_token. Таким образом, тип взаимодействия влияет только на структуру лексического анализатора и функции next_token.

Мы будем разрабатывать параллельное взаимодействие.

1.2. Обнаружение и выделение лексем

Пусть есть некоторый входной поток. Цикл предварительного анализа сканирует его и находит первый значащий символ. Обозначим этот символ cc (current character). Символ cc является левой границей лексемы, и принадлежит ей. Далее запускается функция конечного автомата, которая сканирует входной поток и рассматривает каждый полученный символ cc.

Нам нужно учитывать, что обнаружение правой границы лексем различных типов происходит по-разному. Рассмотрим лексему «целое число». Для простоты принимаем, что текст целого числа состоит только из цифр. Тогда первый символ cc, не равный цифре, уже не принадлежит лексеме, и может являться первым символом следующей лексемы, если, например, входной поток содержит символы 12+34.

Теперь рассмотрим лексему типа «строковый литерал». Пусть первым символом литерала является двойная кавычка. Для простоты примем, что текст литерала не содержит двойных кавычек. Тогда в тот момент, когда символ cc станет равен двойной кавычке, прием лексемы завершается, но символ cc указывает не на первый символ следующей лексемы, а на последний символ текущей. Если после этого управление возвращается в цикл предварительного анализа, то этот символ ошибочно будет указывать на лексему «строковый литерал».

Разные типы лексем могут содержать или не могут содержать лексические ошибки. Рассмотрим лексему типа «идентификатор». Пусть идентификатор — это последовательность букв и цифр, начинающаяся с буквы. Тогда любая не буква и не цифра не принадлежит лексеме и не может вести к ошибочному идентификатору. Справедливости ради заметим, что это не всегда так. Например, в языке Basic идентификатору может быть приписан суффикс в виде одного знака, указывающего на тип переменной. В этом случае распознавание идентификатора может вести к лексической ошибке. То же самое можно сказать о целом числе. Если целое число состоит только из цифр, лексической ошибки возникнуть не может.

Мы условимся, что имя функции конечного автомата, который распознает лексему, не вызывающую лексических ошибок, начинается с префикса get_, а имя функции автомата, распознающего лексему, которая может содержать лексические ошибки, начинается с префикса is_. Например, функция автомата для распознавания идентификатора назовем get_id, а функцию для распознавания строкового литерала назовем is_quote. В практическом языке программирования почти каждая лексема может содержать лексическую ошибку, поэтому это соглашение условное. В рамках практических работ будут действовать максимальные упрощения входного языка, с тем, чтобы работы можно было успешно выполнить за отведенное время.

Замечу, что при практическом создании транслятора никто и никогда не проектирует лексический анализатор. Есть программы типа lex, которые генерируют лексический анализатор автоматически, если на вход программы подать текст с унифицированным описанием лексем. Мы делаем это исключительно ради удовольствия от процесса ☺.

1.3. Рабочее пространство

Все работы выполняются в проекте simplet, шаблон этого проекта предоставляется преподавателем.

Получите архив simplet. Извлеките каталог simplet в корневой каталог диска C:. Откройте проект. Убедитесь, что целевой платформой является x86 или Win32. Проект состоит из следующих модулей:

simplet.cpp — основная функция,
types.h — основные типы,
sttot.h — таблица перекодировки символов в псевдосимволы,
lexan.h — класс лексического анализатора,
sa*nxxx*.h — классы синтаксических анализаторов,
synta*xx*.h — синтаксические управляющие таблицы,
stsyms.h — простая таблица символов,
ststrip.h — классы ленты внутреннего представления.

Рассмотрим структуру модуля simplet.cpp:

```
int simplet(FILE * input) {
    // синтаксический разбор
    return (new syntaxan(
        new lexan(input, stdout), stdout, stdout))->parse();
}

int main(int argc, char * argv[]) {
    FILE * input = fopen("c:\\simplet\\1.st", "rt");
    if (!input) {
        printf("simplet: error while open input file\n");
        return 1;
    }
    return simplet(input) == 0 ? 1 : 0;
}
```

Как видим, входной файл для простоты задается непосредственно.

Если входной поток не удалось открыть, программа завершается неудачно. Если входной поток открыт, вызывается функция транслятора.

Функция транслятора создает синтаксический и лексический анализаторы. Синтаксическому анализатору передается ссылка на лексический анализатор и два потока, для вывода результатов и для вывода сообщений. Лексическому анализатору передается входной поток и поток для сообщений. Метод parse класса syntaxan выполняет синтаксический разбор.

1.3.1. Перечисление символов

Следующий модуль types.h. Здесь заданы константы, ограничивающие длину лексем «идентификатор» и «строковый литерал». Далее определяется тип данных stack_t, используемый для обозначения символов грамматики и транслятора. Затем определяется важнейшее перечисление, в котором задаются символы грамматики (токены и нетерминалы), символы генератора кода. Сейчас в перечислении указаны константы, обозначающие токены, нужные для начала работы. Когда дело дойдет до синтаксического разбора, перечень констант будет определен генератором SYNAX, и тогда изменять что-либо в этом перечислении будет нельзя.

Ниже определена структура токена в минимальном объеме. Токен, — это тип токена stt, целочисленное значение, если есть, вещественное значение, если есть, и строковое представление, если требуется.

В конце модуля определен шаблон класса стека. Он используется в основном как стек МП-автомата. Каких-либо особенностей в нем нет.

1.3.2. Класс синтаксического разбора

Рассмотрим теперь класс syntaxan, модуль sa1null.h. Класс syntaxan этого модуля выводит поток токенов в методе parse. Рассматривая его, видим, как запрашиваются очередные токены и каждый токен выводится при помощи вспомогательной функции print_tok. Когда в потоке окажется токен, обозначаемый константой TOK_EOT, вывод завершается, это конец потока.

Другие модули с названием sa*nxxx*.h также содержат классы syntaxan, в них будут описываться различные анализаторы. Модуль выбирается директивой #include в модуле simplet.cpp. Такая конструкция проекта сокращает объем конкретных модулей, иногда кода будет очень много.

1.3.3. Класс лексического анализатора

Теперь перейдем в класс lexan, модуль lexan.h. В нём определен текущий символ cc и функция next_char, абстрагирующая чтение входа. Затем определяются функции конечных автоматов.

В конце класса находится функция, возвращающая очередной токен входного потока. Эта функция полностью определена, и нужно понять, как она работает.

Прежде всего нужно обратить внимание на то, что первый символ потока принимается в конструкторе. Это совершенно необходимо, потому что функция next_token продолжает работу после работы какого-нибудь автомата в предположении, что символ cc, следующий за очередной лексемой, считан этим автоматом. Поэтому next_token не может начинаться со считывания символа потока, иначе какой-нибудь символ будет потерян.

Сначала функция next_token проверяет текущий символ cc, пытаясь определить, является ли символ значащим. Незначащие символы должны быть пропущены. Поэтому функция начинается с цикла, перебирающего символы потока до тех пор, пока не найдется значащий знак. Делает она это следующим образом.

Если текущий символ равен концу файла EOF, функция формирует токен типа TOK_EOT и завершается.

Если текущий символ равен переводу строки 10, то в зависимости от входного языка, функция либо пропускает этот символ, либо формирует токен типа TOK_LF. В этом месте должен быть также подсчет текущей строки входного файла. Мы ничего подсчитывать не будем для простоты.

Если текущий символ слеш, то в языке Си он может быть как операцией деления, так и началом лексемы «комментарий». Поэтому, если входной язык Си, нужно вызвать автомат для разбора комментария, который решит, что означает слеш. Если автомат «скажет», что это операция, то мы формируем токен типа TOK_DIV и завершаем успешно, возвращая 1. Если автомат «скажет», что в комментарии ошибка (обнаружен конец файла), то мы завершаем разбор, посылая синтаксическому анализатору ноль. Если же автомат успешно разберет комментарий, то токен не формируется, ищем следующий значащий символ.

Если текущий символ имеет код меньше 33, а 32 — это пробел, то мы пропускаем этот символ. Таким образом мы пропустим все пробельные и управляющие символы, так как они не участвуют в процессе.

Наконец, если все эти проверки пройдены, символ значащий, и мы выходим из цикла при помощи break. Если цикл не завершился при помощи break, считываем следующий символ потока. После того, как значащий символ найден, мы разбираем его и вызываем соответствующий автомат.

1.3.4. Таблица перекодировки

Перейдем в модуль sttot.h. В нем находится таблица перекодировки, используемая в анализе для замены символов псевдосимволами. Эта таблица управляет лексическим анализом в том смысле, что в ней задаются допустимые и недопустимые символы входного потока.

Сами псевдосимволы перечислены в начале модуля. Перед началом разработки лексического анализатора эта таблица должна быть правильно настроена. В ней перечислены все 256 возможных символов. Вторая половина таблицы вся помечена константой INVAL, обозначающей недопустимый символ, так как в этой части находятся национальные символы ASCII, которые могут попасть только в комментарий.

Все буквы латинского алфавита помечены константой ALPFA, а все цифры — константой DIGIT. Остается разобраться со знаками операций. Все символы, которыми могут начинаться знаки операций и пунктуаторы, должны быть помечены константой OPERA. Заметим, что вторые символы операций, если они есть, не должны отмечаться, если они не являются одновременно первыми символами.

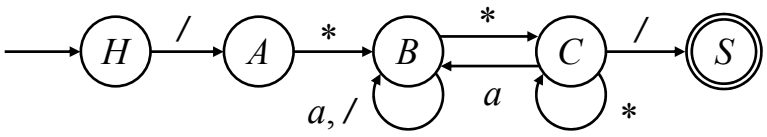
Теперь должно быть понятно, как происходит выбор конечного автомата. Анализируется не символ cc, а псевдо-символ TOT[cc], возвращаемый таблицей перекодировки. Псевдосимволы используются также в конечных автоматах везде, где это возможно.

Если мы усвоили, как устроен данный транслятор, можем приступать к разработке конечных автоматов.

1.4. Разбор лексем

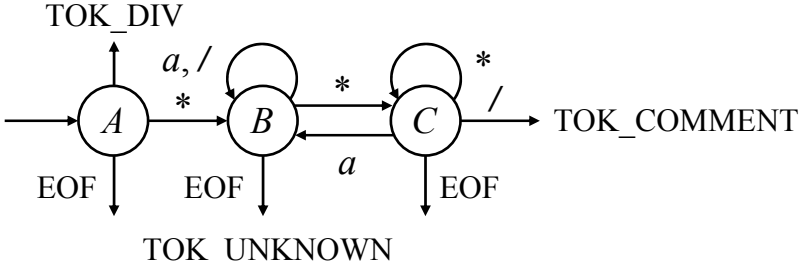
1.4.1. Разбор комментария Си

Граф переходов автомата имеет следующий вид:



Он не учитывает того, что символ, обозначаемый *a*, может быть символом EOF. Кроме того, управление приходит в автомат тогда, когда первый символ комментария уже просканирован, и фактически мы начинаем разбирать, начиная с состояния *A*. Поскольку первый символ комментария совпадает с операцией деления, автомат должен возвращать TOK_DIV в случае, если в состоянии *A* текущий символ не звездочка (и не EOF).

С учетом этого и места расположения вызова функции в программе, мы должны реализовать автомат, граф которого следующий:



Если в любом из состояний обнаруживается конец текста, автомат завершает работу, выводит сообщение типа «\nlexan: unexpected end of file in comment\n\n» в поток ошибок, и возвращает значение TOK_UNKNOWN. Если в состоянии *A* автомат не обнаруживает звездочку, он завершает работу и возвращает значение TOK_DIV. Если в состоянии *C* обнаруживается слеш, автомат завершает работу, возвращая значение TOK_COMMENT.

Переходим в функцию is_comment.

Сначала объявляем перечисление состояний, переменную состояния state и начальное состояние *A* следующим образом:

```
enum {A, B, C} state = A;
```

Затем формируем бесконечный цикл. Первый оператор цикла читает очередной символ потока cc при помощи функции next_char.

После чтения символа формируем оператор switch, разбирающий состояния *A*, *B* и *C*. Далее выполняем анализ текущего символа cc в каждом из состояний. Анализ лучше всего выполнять оператором switch.

Состояние *A*. Если текущий символ равен EOF, выводим сообщение с двумя \n в конце и возвращаем константу TOK_UNKNOWN. Если текущий символ равен звездочке, меняем состояние state на *B*. По умолчанию возвращаем константу TOK_DIV.

Состояние *B*. Если текущий символ равен EOF, выводим сообщение с двумя \n в конце и возвращаем константу TOK_UNKNOWN. Если текущий символ равен звездочке, меняем состояние state на *C*. Все другие символы оставят автомат в состоянии *B*, ничего для этого делать не нужно.

Состояние *C*. Если текущий символ равен EOF, выводим сообщение с двумя \n в конце и возвращаем константу TOK_UNKNOWN. Если текущий символ равен звездочке, то ничего не делаем, просто break. Если текущий символ равен слешу, то возвращаем константу TOK_COMMENT. По умолчанию, когда текущий символ не звездочка и не слеш и не EOF, меняем состояние state на *B*.

Приступаем к отладке. Входной файл содержит только /1, причем буквально ровно два символа, никаких Enter нажимать не надо.

Запускаем программу, анализируем каждый шаг автомата, убеждаемся, что после завершения функции is_comment текущий символ равен 1.

Теперь входной файл содержит /*/*/1. Анализируем каждый шаг автомата, убеждаемся, что после завершения функции is_comment текущий символ равен /, а не 1. При этом, поскольку комментарий пропускается, то символ 1 будет считан в начальном цикле функции next_token.

Далее проверяем следующие входные потоки:

/*/1

/*/*1**/*/1

В первом случае генерируется ошибка, во втором случае после завершения автомата текущим символом должна быть последняя звездочка.

1.4.2. Разбор идентификатора

Функция get_id. Прием идентификатора очень простой, происходит в одном состоянии. Мы входим в автомат, распознав первую букву. За этой буквой уже может ничего не следовать из символов идентификатора, но может следовать, например, знак сложения.

Функция автомата начинается с установки значения TOK_ID в поле stt параметра tok. Затем следует бесконечный цикл. В цикле сначала при помощи оператора switch разбираем псевдосимвол TOT[cc]. Если псевдосимвол равен ALPHA, ничего не делаем, break, если DIGIT, то же самое, break, а по умолчанию возвращаем единицу (успешный разбор). После switch считываем следующий символ при помощи функции next_char.

Для отладки используем следующие тексты: z+3, z1+3, zz21+3.

Во всех случаях после завершения работы функции автомата текущим символом должен быть знак сложения. Если текущий символ равен z, 1 или 2, автомат работает неверно.

1.4.3. Разбор литерала целого числа

Как ранее было сказано, целое число в целях простоты состоит только из цифр. Автомат для приема такого числа очень похож на автомат для приема идентификатора, и даже проще, так как в нем отсутствует анализ псевдосимвола ALPHA. Фактически это самый простой автомат.

Реализуем автомат в функции is_number. В начале функции установим значение TOK_I4 в поле stt параметра tok, так как эта константа обозначает литерал целого числа. Для тестирования используем тексты 1+, 12+. По завершении работы функции автомата символ cc должен быть равен знаку сложения.

1.4.4. Разбор литерала вещественного числа

Вещественные числа являются одними из самых сложных лексем, так как есть множество форм их записи. Мы ограничимся только следующими формами: 0., .0 и 0.0. Здесь ноль условно представляет последовательность цифр, то есть целое число, так, как мы его описали ранее. Иначе говоря, если автомат для разбора целого числа встретит точку, следует принять вещественную часть числа, которая может быть пустой. Если же первый символ лексемы точка, то за ней должна следовать хотя бы одна цифра.

Заметим, что в других языках могут быть другие правила. Например, в языке Pascal вещественное число не может завершаться или начинаться точкой, к тому же точка является признаком конца программы.

С учетом сказанного, автомат для разбора вещественной части числа в нашем случае должен иметь параметр, указывающий, что следует принять хотя бы одну цифру, если число начинается с точки.

В начале функции is_float установим значение TOK_R8 в поле stt параметра tok. Разбор вещественной части происходит примерно так же, как и разбор целого числа. Однако есть следующие отличия.

Во-первых, считываем следующий символ при помощи next_char не после, а до оператора switch, так как точка не формирует значения.

Во-вторых, должен разбираться псевдосимвол CHDOT. Если он будет обнаружен, автомат выводит сообщение «\nlexan: extra dot in float\n\n», и завершает функцию, возвращая ноль. Точка в числе только одна.

В-третьих, если в состоянии по умолчанию параметр must равен единице, то автомат выводит сообщение «\nlexan: single dot in float\n\n», но завершает функцию, возвращая единицу. Таким образом, разбор потока не завершается, но выводится предупреждение. Это пример автоматической коррекции входного потока. Какое-нибудь число все равно будет сформировано, но внимание программиста будет обращено на неверную запись.

В-четвертых, если принята хотя бы одна цифра, параметр must следует переустановить в значение 0.

Для тестирования используем тексты: .+, .1+, .12+, .12.+ и .1.2+.

Убеждаемся, что если автомат завершается успешно, текущий символ является знаком сложения.

Если автомат is_float работает верно, то переходим к редактированию автомата is_number. Добавим в него разбор псевдосимвола CHDOT. Обнаружив точку, автомат is_number возвращает вызов автомата is_float со вторым параметром, равным нулю. Для тестирования используем те же тексты, что и для is_float, с добавлением единицы в начало каждого текста.

1.4.5. Разбор строкового литерала

Для простоты примем, что строковый литерал может содержать любые символы, код которых больше или равен 32 (пробел), но символ если частью литерала является символ двойной кавычки, являющийся границей лексемы, то он должен повторяться два раза. Такой способ используется, например, в языке Basic. Заметим, что граничные символы двойной кавычки не входят в лексему, их пропускают при формировании текста токена.

Функция is_quote начинается с установки значения TOK_QUOTE в поле stt параметра tok. Далее формируем бесконечный цикл. В цикле сначала считываем очередной символ входа при помощи функции next_char. Затем проверяем символ и псевдосимвол при помощи условного оператора.

1) если псевдосимвол равен QUOTE, то кавычка либо завершает литерал, либо является началом двух кавычек. Поэтому считываем следующий символ при помощи next_char, и проверяем псевдосимвол. Если псевдосимвол не равен QUOTE, то кавычка завершает литерал, следующий символ принадлежит следующей лексеме, завершаем функцию, возвращаем единицу. Если псевдосимвол равен QUOTE, то кавычка часть литерала.

2) если символ равен EOF, формируем сообщение «\nlexan: unexpected end of file in quote\n\n», завершаем функцию, возвращаем ноль.

3) если код символа меньше 32 (пробел), символ не допустим, формируем сообщение «\nlexan: extra character in quote\n\n», завершаем функцию, возвращаем ноль.

4) если ни одно из перечисленных, ничего не делаем, это символ литерала, впоследствии мы сформируем сам литерал.

Тестирование проводим на следующих текстах: ", "", """, """", "z", "я", "1", "-", "Enter", где Enter — нажатие клавиши Enter.

1.4.6. Разбор операций

Если лексическому анализатору не встретилась ни одна из разобранных выше лексем, значит текущий символ cc является началом символа операции, знака пунктуации или недопустимым символом. Все эти случаи разбирает функция is_opera.

Заметим, что символ операции может состоять из одного, двух и даже трех знаков, например, ~, <>, !=, !==. Поэтому анализируем текущий знак cc при помощи оператора switch. Если символ совпадает с одним из знаков однознаковой операции или пунктуатора, то считываем следующий символ входа, записываем в поле stt параметра tok константу операции, завершаем функцию, возвращаем единицу. Если символ cc совпадает с одним из первых знаков многознаковой операции, то считываем следующий символ, и сравниваем его со втором знаком операции. Если символ cc совпадает со вторым знаком операции, то операция двухзнаковая, тогда считываем еще один символ входа, записываем в токен константу двухзнаковой операции и завершаем функцию, возвращая единицу. Если символ cc не совпадает со вторым символом операции, то записываем в токен константу однознаковой операции, и завершаем функцию, возвращая единицу.

Если символ cc не является ни операцией, ни пунктуатором, выводим сообщение «\nlexan: extra character in line\n\n», возвращаем ноль.

Единственная сложность здесь — правильно выбрать константу. Так, TOK_ASS обозначает операцию «присвоить», TOK_EQ — «равно», TOK_NE «не равно», TOK_LT <, TOK_GT >, TOK_LE <=, TOK_GE >=, TOK_LP (, TOK_RP), TOK_LS {, TOK_RS }, TOK_ADD +, TOK_SUB –, TOK_MUL *, TOK_DIV /. Другие константы можно узнать в программе SYNAX, введя в нее грамматику из одного правила *A*→знак.

Для тестирования используем текст вида: 1+a–1*z/1<2>3==4…

1.5. Контрольные вопросы и упражнения

1. Назовите все виды лексем, охарактеризуйте каждый вид, опишите обнаружение и выделение лексемы каждого вида.

2. Работа ST-102. Поток токенов

Цели:
- формирование атрибутов токенов.
Задачи:
- формирование текста токена;
- формирование целого значения токена;
- формирование вещественного значения токена;
- обнаружение ключевых слов.
Опорные документы:
[1]

2.1. Атрибуты токена

Сам по себе токен, состоящий только из типа токена stt, позволяет выполнить анализ входного потока и определить цепочку правил. Для целей генерации выходного текста или выполнения непосредственно вычислений такой токен непригоден. Поэтому токен — это структура данных, которая содержит некоторую дополнительную информацию.

1. Если токен является идентификатором, то существенно необходимо знать текст этого токена, так как этот текст позволяет отличить один идентификатор от другого.

2. Если токен является числом, существенно необходимо знать, какое это число, иначе невозможно будет выполнить вычисления. То же самое можно сказать обо всех литеральных значениях (константах).

3. Некоторые идентификаторы являются ключевыми словами языка. Есть два подхода к ключевым словам: а) они либо идентификаторы, б) либо самостоятельные токены, обозначающие конкретное ключевое слово. Мы придерживаемся второго подхода, так как он представляется гораздо более удобным для последующего синтаксического анализа.

4. Для формирования сообщений об ошибках токен должен содержать информацию о своем местоположении в тексте, а именно, номер строки и позицию в строке.

Поэтому лексический анализатор должен не только обнаруживать и выделять лексемы, но и записывать в структуру токена перечисленную выше дополнительную информацию. Это и есть цель данной работы. В целях простоты мы не формируем позицию токена, хотя это не сложно.

2.2. Вычисление текста токена

Формировать текст токена необходимо только для идентификаторов и строковых литералов. Для токенов других видов текст токена не является обязательным, и его можно либо формировать, либо не формировать.

Рассмотрим текст идентификатора. Поскольку язык идентификатора является бесконечным, идентификатор, в принципе, может иметь какую угодно длину. Однако есть два соображения, заставляющие нас ограничивать фантазию программиста.

1. Длина строки программного кода не может быть безграничной по чисто техническим причинам, а также потому, что длинные строки сложно отображать и, что еще хуже, сложно читать.

2. Идентификаторы в процессе трансляции заносятся в базу данных трансляции, называемую таблицей символов. Чем длиннее идентификатор, тем большее время необходимо для его добавления и поиска. Это ограничение куда более существенно, чем предыдущее, потому что оно прямым образом сказывается на скорости трансляции.

Поэтому в трансляторах существует понятие *распознаваемой длины идентификатора*. Программист может выдумать идентификатор какой угодно длины, однако любой транслятор ограничит её, формируя токен. В этом несложно убедиться. Например, компилятор Borland C++ распространенной версии 3.51 ограничивает длину идентификатора 32-мя знаками, а в пакетных файлах MS-DOS 6 эта длина составляет 8 знаков. В компиляторе Microsoft Visual C++ распознаваемая длина равна 247 знаков.

Если рассматривать токен строкового литерала, то здесь ограничение длины связано в основном с допускаемой длиной строки, хотя многие языки позволяют записывать строковые литералы на нескольких строчках. В языке PHP, например, длина строкового значения ничем не ограничена, и литерал может состоять из любого числа строк (ограничение существует всегда, и оно связано с объемом доступной оперативной памяти, и, хотя можно воспользоваться дисковой памятью, она тоже не бесконечна).

В нашем трансляторе есть два ограничения, на длину идентификатора MAX_ID, и на длину литерала MAX_QOUTE. Заметим, что метод append токена добавляет в строковое значение токена не более чем MAX_QOUTE знаков. Поэтому нам остается только ограничить длину идентификатора

Есть еще одна весьма важная деталь, касающаяся текста. Некоторые языки являются регистро-зависимыми, как, например, Си. Для Си текст не подвергается никаким преобразованиям. Другие языки являются регистронезависимыми, как, например, Basic или Pascal. Для этих языков текст идентификатора требует преобразования либо в нижний, либо в верхний регистр. Текст строкового литерала никогда не изменяется, если только он не содержит управляющих последовательностей.

Буква переводится в верхний регистр операциями И НЕ 32, где операции битовые, а не логические. Буква переводится в нижний регистр битовой операцией ИЛИ 32. Например, пусть буква равна *A* латинская, код 0100001. Значение 32 (пробел) соответствует пятому разряду, который равен нулю. Операция НЕ 32 возвращает 11011111. Операция И этих кодов дает код 0100001, *A*. Операция ИЛИ 32 дает код 01100001, то есть *a*.

Важно также, что операции И НЕ 32 изменяют код цифр, а операция ИЛИ 32 не изменяет его. Поэтому операцией ИЛИ 32 можно изменять и буквы и цифры. Код цифры 0 равен 00100000. Код знака подчеркивания 01011111, поэтому он изменяется операцией ИЛИ 32, но не изменяется операциями И НЕ 32. На практике регистро-независимые языки обычно выполняют перевод в верхний регистр, например, в ассемблерах, в Basic или в MS-DOS. Пока мы не будем делать перевод, оставим это до момента, когда станет известен язык.

Переходим в метод get_id. Описываем добавление символа cc методом append токена. Для тестирования используем текст abc+xyz. Убеждаемся, что идентификаторы не теряют букв. После этого ограничиваем длину идентификатора константой MAX_ID. Для этого нужно вычислить длину записанного идентификатора, и если она меньше MAX_ID, то добавляем символ, иначе ничего не делаем.

Переходим в метод is_quote. Описываем добавление символов методом append токена, в двух местах, когда символ вторая кавычка, и когда символ допустим после всех проверок. Тестируем особенно тщательно, добавляя в литерал одну, две, три кавычки, в соседстве и порознь.

2.3. Выявление ключевых слов

Ключевое слово — это зарезервированный идентификатор, поэтому при непрямом лексическом анализе оно определяется после определения идентификатора. Для этого нужно немного изменить код метода get_id.

В том месте оператора switch, где возвращается единица (default), вместо оператора return 1 нужно вписать оператор return is_keyword. Тогда метод is_keyword должен возвращать единицу и устанавливать поле stt токена либо в значение константы ключевого слова, либо в значение TOK_ID.

Общий вид метода примерно следующий:

```
int is_keyword(sttoken & tok) {
    if (!strcmp(tok.str_val, "print")) {
        tok.stt = TOK_PRINT;
        return 1;
    }
    if (!strcmp(tok.str_val, "if")) {
        tok.stt = TOK_IF;
        return 1;
    }
    . . .
    return 1;
}
```

Ключевые слова print, if, else будут во всех языках. Другие ключевые слова будут зависеть языка. Сейчас можно вписать несколько слов, потом, когда дело дойдет до генератора SYNAX, вписывать нужно будет ровно те слова, которые сгенерирует генератор, и никакие другие.

Тестируем на разных идентификаторах.

2.4. Вычисление целочисленного значения

Целочисленное значение вычисляется для целочисленных и вещественных литералов. Делается это, в принципе, легко, однако есть одно но.

Ради интереса напишите в начале функции main следующий код:

```
int main(int argc, char * argv[]) {
    int x = 4294967297;
    double y = 4294967297;
    . . .
}
```

Исполните эти две строчки и посмотрите, чему равны x и y. Заметим, что число 4294967297 превышает диапазон физического представления числа типа int ровно на единицу. Возможно, что ваш компилятор воспротивится, компиляторы становятся все «умнее». Но в старых компиляторах Си значение *x* будет равно единице. Язык целого числа бесконечен, и программист может написать литерал числа любой длины, и возникает вопрос, как к этому должен относиться компилятор?

Разные компиляторы относятся к этому по-разному. Компилятор языка Pascal или Basic скажет, что константа недопустима. Языку Си все равно, он из недопустимого числа сделает допустимое, правда, это будет уже совсем другое число. Мы пока поступим как Си.

Для вычисления целого значения нужен накопитель, изначально равный нулю. Пусть литерал равен "235". Первый символ cc равен '2'. Умножаем накопитель на 10, прибавляем cc и вычитаем символ '0'. Вычитая из символа '2' символ '0', получаем вместо кода цифры собственно число 2, потому что таблицы символов устроены так, что разница кодов цифры N и цифры 0 в точности равна значению цифры N. Тогда в накопителе будет записано число 2. Далее поступаем точно так же. Умножаем 2 на 10, получаем 20, прибавляем значение цифры 3, получаем 23. Умножаем 23 на 10, прибавляем значение цифры 5, получаем 235, и это то, что нужно.

Переходим в метод is_number и описываем эти действия, считая накопителем поле int_val токена. Этот накопитель уже равен нулю, токен был нами подготовлен заранее. Заметим, что никаких дополнительных переменных не требуется, все выполняется двумя операциями *= и += над накопителем. Тестируем полученный код на разных числах, в том числе и на числе $2^{32}+1 = 4294967297$. Если при этом значение не равно единице, что-то неверно в трансляторе.

Как бы было хорошо, если бы на этом всё. Увы, только в нашем трансляторе на этом всё, в реальном трансляторе на этом все только начинается. Потому что в большинстве языков недопустимые значения числовых литералов пресекаются, как в Pascal. Тогда нам нужен другой подход. Само по себе значение вычисляется так же, но нам нужно знать, не превышает ли значение диапазон представления.

И сразу возникает вопрос, а диапазон какого представления? Программист ведь может написать:

```
char c = 4294967297;
```

С точки зрения теории, программист — пользователь, а любой пользователь по определению идиот (то есть человек, мыслящий иначе). Нужно понять одну важную вещь, — лексический анализатор выявляет лексемы, и больше ничего. Он не может знать, что числовой литерал присваивается переменной определенного типа. Поэтому все числовые константы считаются константами типа general, наибольшего допустимого в языке (если явно не сказано иное). Если значение превышает диапазон наиболее возможного представления, тогда можно говорить об ошибке.

Можно ли определить, что значение литерала превышает диапазон представления? Легко. Сначала нужно определиться с диапазоном представления, после чего взять накопитель большего размера. Например, если наибольшее целое число в языке имеет размер 32 бита, то нужно взять накопитель размером 64 бита, в Microsoft Visual C это тип __int64. Тогда, как только в пятом (считая от единицы) байте накопителя появится какое-либо значение, накапливаемое число вышло за диапазон представления. Это легко определяется битовой операцией И с константой 0xFFFF0000. Если в языке транслятора нет типа с бо́льшим представлением, то возникает проблема, которую решают при помощи ассемблера.

Предположим, мы определили, что число вышло за диапазон представления. Но есть еще одно но. Число, входящее в диапазон представления, может оказаться настолько большим, что стать отрицательным. И тогда непонятно, является ли число допустимым. Потому что лексический анализатор должен быть очень сложным для того, чтобы распознавать унарный минус, который может предшествовать числу. А без знания того, представляет число положительное или отрицательное значение, невозможно понять, допустимо ли значение.

Поэтому с этими проблемами лучше разбираться на стадии синтаксического анализа, когда последовательность лексем станет известной. В этом случае унарный минус (или плюс) — часть синтаксической, а не лексической грамматики. Синтаксический унарный минус (или плюс) гораздо проще, хотя иногда он приводит к проблеме (язык не может быть детерминировано распознан определенным синтаксическим анализатором).

Это еще не всё. Некоторые числовые литералы являются восьмеричными, шестнадцатеричными или двоичными. С ними, правда, проще, потому что основания их систем счисления кратны степени двойки и можно заменить операцию умножения (на десять) операцией сдвига. С этой точки зрения получение значения литерала получается проще. И мы еще совсем не рассматриваем случай, когда числовой литерал имеет суффикс, определяющий тип значения, хотя на практике это встречается часто.

Заметим также, что ничто не мешает при параллельной работе синтаксическому анализатору сообщать функции next_token лексического анализатора, какой собственно токен ожидается, и так делают. Тогда легко решается проблема с унарным минусом, поскольку синтаксический анализатор его распознает раньше значения.

2.5. Вычисление вещественного значения

С вычислением значения вещественного литерала проблем тоже немало, но мы ограничились определенными формами записи, часть проблем исчезла. Будем вычислять вещественное значение, вычисляя целую часть числа как описано выше, и это сделает функция is_number, а затем прибавляя к ней вещественную часть, и это сделает функция is_float.

Вещественная часть числа выражается целым числом, как это следует из конструкции функции is_float. Нам нужен накопитель и счетчик чисел после десятичного разделителя. Объявим в функции is_float переменную number типа double и переменную div типа int. Вычисляем целое значение дробной части точно также, как это происходит в функции is_number, но при каждом действии накопления подсчитываем цифру в div. Тогда в тот момент, когда в default возвращается единица, записываем в поле dbl_val значение поля int_val, к которому прибавим накопитель number, поделенный на 10^{div} (что может быть вычислено при помощи функции pow). Описываем эти действия, и тестируем на разных вещественных литералах.

2.6. Контрольные вопросы и упражнения

3. Работа ST-103. Рекурсивный спуск

Цели:
- исследование метода рекурсивного спуска.
Задачи:
- расширенное применение метода рекурсивного спуска.
Опорные документы:
[1]

3.1. Метод рекурсивного спуска

Рекурсивный спуск является простым методом, в котором разбор всех правил одного нетерминала выполняет процедура одна процедура. Метод достаточно подробно описан в [1]. Реализуемая грамматика следующая:

$E \rightarrow E+T \mid E-T \mid T$
$T \rightarrow T*P \mid T/P \mid P$
$P \rightarrow num \mid (E)$

Здесь num обозначает число, которое может быть как целым, так и вещественным. Токены — это число, четыре знака операций и скобки.

Целью работы является построение синтаксического анализатора, который вычислит входное выражение со скобками.

3.1. Рабочее пространство

Работа выполняется в модуле sa2RD.h, который нужно подключить в модуле simplet.cpp, отключив модуль sa1null.h.

3.3. Контрольные вопросы и упражнения

4. Работа ST-104. Синтаксические грамматики

Цели:
- разработка синтаксических грамматик.
Задачи:
- грамматика для выражений;
- грамматика операторов;
- грамматика модуля.

Опорные документы:
[]

4.1. Рабочее пространство

Для выполнения последующих работ требуется разработка синтаксических управляющих таблиц. Эту сложную задачу выполнит генератор SYNAX. Управляющая таблица строится на основе грамматики, поэтому сначала нужно разработать грамматику на основе одного из базовых языков (Basic, Pascal или Си), и язык нужно выбрать в самом начале работы. Разрабатывается две грамматики, одна для метода разбора LL(1), вторая для метода SLR(1) (или LALR, если вдруг грамматика не SLR).

Сначала в окно генератора нужно ввести грамматику, сохранить ее в файл и выбрать в меню «Анализ». Если анализ грамматики успешен, далее нужно выбрать вкладку метода разбора, и снова выбрать в меню «Анализ». Если анализ грамматики для данного метода разбора успешен, можно выполнить экспорт из вкладки метода разбора и из вкладки Symbols, выбирая в меню «Экспорт».

Из вкладки символов генерируется текст, содержащий константы перечисления sttype. Часть этого файла вставляется непосредственно в проект, в модуль types.h. Порядок перечисления не может быть изменен, так как на его основе строится управляющая таблица. При этом нужно следить за тем, чтобы перечисление начиналось следующим образом:

```
typedef enum sttype : stack_t {
    OUT_START = 997,
    TOK_COMMENT,
    TOK_UNKNOWN,
    // токены
    TOK_EOT = 0,
```

Иначе говоря, начальная часть перечисления не должна меняться. Неиспользуемые константы можно располагать в конце перечисления для обеспечения целостности программного кода. Из вкладки метода разбора генерируется управляющая таблица со структурами данных. Сгенерированный текст как есть вставляется в модуль вида synta*xx*.h, в зависимости от задания. В этой работе ничего никуда не вставляем.

4.2. Выбор языка

Сначала нужно определиться с языком. В качестве базового берем один из распространенных языков программирования, такой, как Basic, и определяем его подмножество, ограничивая набор типов данных, операций и операторов. Для исследования метода разбора можно ограничиться одним типом данных, но при этом некоторые моменты окажутся «за бортом», поэтому в предлагаемых вариантах два типа данных. Перечень операций желательно иметь как можно меньше, но это сильно обедняет язык, поэтому варианты предлагают два компромисса. Среди операторов нет цикла.

Возможные перечни типов:

- вариант T1: целый строковый
- вариант T2: вещественный строковый

Возможные перечни операций:

- вариант O1: + – * / < <= > >= равно И
- вариант O2: + – * / < > равно И ИЛИ НЕ

Возможные перечни операторов управления:

- вариант S1: объявление присвоение if print составной пустой

Кроме того, язык регистро- независимый РН или зависимый РЗ.

Тогда возможные варианты языков следующие:

1. Basic РН T1 O1 S1
2. Basic РН T2 O1 S1
3. Basic РН T1 O2 S1
4. Basic РН T2 O2 S1
5. Pascal РН T1 O1 S1
6. Pascal РН T2 O1 S1
7. Pascal РН T1 O2 S1
8. Pascal РН T2 O2 S1
9. Си РЗ T1 O1 S1
10. Си РЗ T2 O1 S1
11. Си РЗ T1 O2 S1
12. Си РЗ T2 O2 S1

Вариант следует согласовать с преподавателем.

4.3. Грамматика для выражений

Грамматики разрабатываем по частям, сначала выражения, затем операторы (управления), затем программный модуль в целом. За основу берем право рекурсивную грамматику G3 для метода разбора LL, и лево рекурсивную грамматику G1 для метода LR. Названия файлов выбираем в формате LL-Basic-Part-N, где Part — часть E, S или M, N — номер версии.

LL-грамматика не содержит левой рекурсии, но имеет вырождающиеся нетерминалы. Условимся обозначать их точкой в конце идентификатора нетерминала. LR-грамматика *не имеет* вырождающихся нетерминалов.

Тогда исходная LL-грамматика выражения имеет вид:

E→ TT.
T.→ +TT. | –TT.
T.→ λ
T→ PP.
P.→ *PP. | /PP.
P.→ λ
P→ I4 | id | (E)

Целевой символ E. Заметим, что I4 обозначает целочисленную константу, это терминал, идентификатор которого равен в точности [I4], id обозначает идентификатор, это терминал, идентификатор которого равен [id] в формате SYNAX. Вместо λ пишем точку, вместо знака + пишем [+], вместо знака * пишем [*] и т.д. Если заданы вещественные константы, пишем [R8], если заданы строковые литералы, пишем [QUOTE]. Если задано два типа, то пишем два типа констант. Убеждаемся, что грамматика LL(1), сначала анализируя грамматику (это нужно делать при любом ее изменении), затем анализируя метод разбора LL(1).

Теперь нужно внести уровень унарной операции «минус» для записи отрицательных значений. Он располагается между уровнями нетерминалов T и P:

T→ UU.
U.→ *UU. | /UU.
U.→ λ
U→ P | –P
P→ I4 | R8 | id | (E)

Снова анализируем грамматику, затем метод разбора. Если неудачно, то возвращаем грамматику назад, унарного минуса не будет.

Далее открываем методичку по курсовому проектированию и находим в ней приоритеты операторов. Добавляем в грамматику новые уровни в соответствии с приоритетом. Высший приоритет в грамматике имеет нетерминал P, низший — нетерминал E. Для обозначения нетерминалов уровней используем идентификаторы: LAND для уровня, содержащего операции И, LOR для уровня, содержащего операции ИЛИ, LNOT для уровня с операциями логического отрицания, R для уровня, описывающего операции отношения (меньше, больше, равно), например:

E→ RR.
R.→ <RR. | >RR. | =RR. | <>RR.
R→ TT.
T.→ +TT. | –TT.

Здесь наименьший приоритет имеют операции отношения, поэтому отношения в грамматике идут первыми.

После того, как LL-грамматика для выражений будет подготовлена, формируем LR-грамматику для выражений, используя в качестве основы грамматику G1. Это немного попроще, грамматика более понятна.

4.4. Грамматика операторов

Нетерминал, описывающий выражение, обозначим S. Он становится целевым символом, и из него должны выводиться все заданные операторы.

Пустой оператор в Pascal и Си — это точка с запятой. В Basic пустой оператор — это терминал LF (конец строки) Поэтому лексический анализатор для Basic должен выделять этот токен. Кроме того, каждый оператор или каждая строка оператора Basic также завершается LF.

Разрабатываем грамматику последовательно, начиная с простых операторов, переходя к условному оператору в конце.

Сначала добавляем в грамматику пустой оператор.

Убеждаемся, что грамматика LL(1).

4.4.1. Оператор присваивания:

Форматы этого оператора во всех языках примерно одинаковы:

Basic: id = E LF

Pascal: id := E ;

Си: id = E ;

В операторах будет встречаться выражение E. Чтобы упростить построение грамматики, временно введем в нее правило E→*e*, где *e* — терминал-заглушка. Буквально, в формате SYNAX, <E>=[e], и это правило должно быть последним в грамматике. Впоследствии его можно будет заменить грамматикой для выражения.

Добавляем в грамматику оператор присваивания.

Убеждаемся, что грамматика LL(1).

4.4.2. Оператор print

Формат оператора print одинаков для всех:

print E

4.4.3. Условный оператор

Добавляем в грамматику условный оператор. Между языками Basic и Pascal и Си здесь возникает большая разница.

В Basic нет составного оператора и нет никаких скобок, ограничивающих последовательность операторов. В Basic составной оператор начинается с новой строки и каждый следующий оператор с новой строки. Поэтому составной оператор — это форма записи условного оператора или оператора цикла такая, в которой операторы идут каждый с новой строки.

Это означает, что после then E или do E следует LF, после else следует LF, после end if или end do следует LF, в конце каждого оператора LF.

В Pascal и Си в грамматике условного оператора стоит один оператор, но он может быть составным. Формат составного оператора:

Pascal: begin операторы end ;

Си: { операторы }

Поэтому нужно ввести нетерминал SS (операторы), который в Basic вставляется после терминалов then LF и else LF.

Basic

```
if E then LF
    SS
end if LF
```

```
if E then LF
    SS
else LF
    SS
end if LF
```

Pascal

```
if E then S
if E then S else S
```

Си

```
if (E) S
if (E) S else S
```

Из нетерминала SS выводится S или S S (не SS, а два подряд идущих нетерминала S). Но грамматика при этом не является LL. Поэтому SS заменяется на S S., где S. вырождается: SS→ S S., S.→ S S. | λ. В языках Pascal и Си примерно так же описывается нетерминал S (оператор), который может быть составным оператором:

S→ { S S. } , S.→ S S. | λ для Си, или

S→ begin S S. end ; , S.→ S S. | λ для Pascal.

Поскольку будет еще уровень модуля, лучше описать не составной оператор, а нетерминал SS, как в Basic, и добавить правило S→SS, где SS — это один нетерминал. При этом, возможно, грамматика выйдет из LL(1).

Заметим, что при этом составной оператор не может быть пустым. Чтобы он был пустым, нужно S→λ, но это, скорее всего, приведет к проблемам в методе LL(1), но попробовать стоит.

И эта проблема не единственная. Заметим, что в Си в составной оператор может входить любой оператор, в том числе оператор объявления, но только в его начале. А вот в Pascal нет, операторы объявления находятся в начале программы. В Basic оператор объявления может встречаться вообще где угодно.

Мы поступим следующим образом. Все операторы объявления переменных во всех языках будут располагаться только в начале модуля.

Есть еще одна проблема. Поскольку условный оператор предполагает две формы, с else и без else, грамматика становится не LL(1), нужна левая факторизация. Кроме того, эта грамматика неоднозначна! Поэтому мы поступим следующим образом — никаких else. Тогда остается только разобраться с составным оператором. Не до других проблем сейчас. Дополнительно можно почитать описание этого задания в методичке 2017 года, она подробнее, спросите ее у преподавателя, если на сайте ее нет.

4.5. Грамматика модуля

Сохраните грамматику операторов, затем сохраните ее как грамматику модуля, заменим букву S на букву M в названии файла.

Модуль будем понимать следующим образом: это последовательность операторов объявления, за которой следует последовательность других операторов. Тогда M→ DS SS, где DS — последовательность операторов объявления, а SS описано раньше.

Следовательно, нужно описать DS. А для этого нужно описать одноединственное объявление. Для разных языков синтаксис такой:

Basic: dim id as integer , id as integer LF

Pascal: var id , id : integer ;

Си: int id , id ;

Тогда, для примера, в языке Basic:

D→ dim id as integer L. LF

L.→ , id as integer L.

L.→ λ

Тогда DS→ D D., D.→ D D. | λ.

Все бы хорошо, но получается, что ни DS, ни SS не могут быть пустыми. Особенно плохо с DD. Программа не может не содержать оператора действия, но может не содержать оператора объявления, например, программа из единственного оператора print.

Вам стоит попробовать сделать DS вырождающимся (или сделать вырождающимся S, что лучше). Не получится для LL(1), ну и ладно.

Грамматика для языка Pascal строится иначе.

В начале модуля располагаем операторы объявления переменных, а в конце модуля — составной оператор begin..end, завершающийся точкой. В связи с этим точка не может быть началом никакой лексемы, и в массиве TOT она должна быть помечена как INVAL.

После того, как получится построить грамматику для LL(1), нужно сформировать грамматику для SLR(1). Это гораздо проще. Во-первых, никаких вырождающихся нетерминалов. Вырождающаяся последовательность типа DS строится легко следующим образом:

M→ SS | DS SS

Аналогичным образом составной оператор (пример для Си):

S→ {} | { SS }

Здесь левая факторизация не требуется, и практически все, что вы сможете придумать, будет SLR(1). Но при этом не удастся построить грамматику, которая порождает пустую цепочку, в отличие от LL(1), в которой пустые правила допускаются.

4.6. Соединение грамматик

Остается присоединить грамматику для выражения к грамматике для модуля, заменив правило E→*e*. Далее убеждаемся, что грамматика является грамматикой LL(1) или SLR(1).

4.7. Контрольные вопросы и упражнения

5. Работа ST-105. Предиктивный анализатор

Цели:
- построение предиктивного анализатора.
Задачи:
- построение управляющей таблицы;
- разработка алгоритма предиктивного анализа.
Опорные документы:
[1]

5.1. Рабочее пространство

В модуле simplet.cpp нужно подключить модуль синтаксического анализатора LL(1), sa3LL.h, другие модули синтаксических анализаторов должны быть отключены (закомментированы). Модуль sa3LL.h включает в себя модуль управляющей таблицы syntall.h.

5.2. Генерация файлов SYNAX

Открываем SYNAX и грамматику типа LL. Анализируем грамматику, переходим на вкладку Symbols, выполняем анализ и экспорт. Ищем файл экспорта, в его названии присутствует название грамматики и symbols.

Копируем в файле экспорта константы перечисления sttype и вставляем в перечисление в модуле types.h. Следим за тем, чтобы перечисление начиналось следующим образом:

```
typedef enum sttype : stack_t {
    OUT_START = 997,
    TOK_COMMENT,
    TOK_UNKNOWN,
    // токены
    TOK_EOT = 0,
```

Никакие отклонения от порядка следования констант недопустимы, так как константы являются номерами столбцов и строк управляющей таблицы. Изменили грамматику по каким-то причинам, все начинаем заново.

После этого ваш проект, скорее всего, расстроится, потому что некоторые константы продублируются. Все дубликаты нужно найти и удалить. При этом удалять нужно из хвоста перечисления, только после слов «конец символов грамматики». После этих слов, вообще говоря, в дальнейшем будут следовать константы генератора кода, начинающиеся с OUT_. Порядок этих констант неважен, равно как и нахождение за словами «конец символов грамматики» каких-либо других констант. Некоторые константы не требуются для конкретного синтаксического анализатора и остаются в перечислении исключительно для того, чтобы не пришлось заодно править функции, которые выводят поток лексем.

В SYNAX переходим на вкладку LL(1), анализируем и экспортируем. Находим файл экспорта и вставляем его целиком в модуль syntall.h. Начало модуля при этом будем иметь примерно следующий вид:

```
// 2019 ReVoL Primer Template
// syntall.h
// синтаксическая таблица LL

/* Generated by 2016 ReVoL SYNAX */
/* 00-00-2000 00:00:00 */

#define ST_MAX_RULE_LEN 32 /* Max rule length */
```

Константа ST_MAX_RULE_LEN задает максимальную длину правила и размер управляющей таблицы. Далее в модуле расположена таблица правил грамматики:

```
/* Rules */
sttype RULE[29][ST_MAX_RULE_LEN] = {
    /* 0*/{ TOK_EOT },
    /* 1 SYM_M  */{ SYM_S, SYM_M, TOK_EOT },
```

Каждое правило пронумеровано, в комментарии в начале правила указан левый нетерминал. Тело каждого правила заключено в фигурные скобки, каждое правило заканчивается константой TOK_EOT, имеющей нулевое значение, так что к правилу можно применять строковые функции.

Далее в модуле расположена константа MAX_RULE, задающая максимальный индекс правила в массиве.

Далее расположена вспомогательная таблица длин правил:

```
/* Rule Length */
int RLEN[] = { 0,2,0,1,5,3,0,4,6,2,4, . . . };
```

Далее расположена вспомогательная функция для уточнения таблицы RLEN. При построении трансляционной грамматики в правила вручную будут вставляться операционные символы, после чего таблица RLEN станет недействительной. В данной работе функция не требуется.

Далее определены три константы:

```
#define ACC 255 /* ACCEPT CODE */
#define POP 254 /* POP CODE */
#define START SYM_M /* START SYMBOL */
```

Константа ACC обозначает ячейку, допускающую цепочку. Константа POP обозначает действие «выброс». Константа START обозначает целевой символ грамматики, который перед началом работы МП-автомата должен быть помещен на стек.

Далее в модуле следует собственно управляющая таблица SYNTA. Она документирована, ориентироваться в ней легко. Заметим, что если грамматика не LL(1) и таблица сгенерирована, в нее автоматически будет внесена синтаксическая ошибка в месте неоднозначности (или в нескольких местах), предотвращающая компиляцию.

5.3. Разработка алгоритма LL(1)

Если проект компилируется, переходим к разработке модулю синтаксического анализатора LL(1), sa3LL.h.

Требуемые структуры данных уровня класса:

- элемент данных tok типа sttoken. Это текущий токен потока.

- элемент данных sta типа ststack, параметризованный типом stack_t и константой MAX_STACK. Это рабочий стек МП-автомата.

- функция-элемент next_token:

```
// возвращает очередной токен
int next_token() {
    return lex->next_token(tok);
}
```

Метод parse. Он начинается с получения очередного (первого) токена при помощи метода next_token. Если next_token завершается неудачно, то функция завершается и возвращается 0.

Далее идут элементы данных, требуемые для работы МП-автомата. По большей части назначение этих элементов вспомогательное:

- две переменных типа int, *i* и *n*. Это итератор и длина правила.

- два вспомогательных символа at и bt типа stack_t. Они предназначены для анализа работы в случае, если что-то не так. Для нормального функционирования они не нужны. Начальное значение равно TOK_EOT.

- переменная s типа stack_t. Это символ на стеке, начальное значение равно TOK_EOT.

- переменная t типа stack_t, начальное значение TOK_EOT. Это значение, считанное из управляющей таблицы.

Далее идут предварительные действия:

- вызов функции get_rule_len. Она уточняет длину правил, в этой работе она не требуется, но потом мы этот код будем копировать, поэтому не помешает.

- проталкивание на рабочий стек константы TOK_EOT (дно стека).

- проталкивание на рабочий стек константы START (то есть аксиомы грамматики).

Далее следует бесконечный рабочий цикл автомата LL. В цикле автомат должен выполнить следующие действия:

1) получить символ на вершине стека в s;

2) получить в переменную t значение в таблице, используя значение s как первый индекс, а значение tok.stt как второй индекс;

3) анализируя значение t, выполнить соответствующее действие. Это сделает оператор if, который мы рассмотрим далее.

После цикла в поток parse_stream выводится success и возвращается 1:

```
    fprintf(parse_stream, "\nsyntaxan: success.\n\n");
    return 1;
```

Остается разбор значения в таблице t. Описываем оператор if в точном соответствии с теорией (со значениями в управляющей таблице). Заметим, что нулями в таблице обозначены ошибочные состояния автомата, которые никак не анализируются. При обнаружении нуля работа автомата просто завершается с выводом сообщения. Числами в таблице обозначены номера правил, которые нужно применить, при этом число, равное ACC, обозначает допуск, число, равное POP, обозначает выброс. Эти числа не пересекаются с номерами правил. Заметим, что отрицательными значениями можно обозначать ошибочные ситуации, но делать это нужно в реальном трансляторе.

В учебных целях управляющая таблица построена в соответствии с теорией. Фактически первая ее часть, содержащая константы ACC и POP, может быть смоделирована алгоритмически, при этом таблица может быть значительно сокращена.

Условия оператора if (пишем только комментарий, действия потом):

- если t меньше либо равно 0, это ошибка;
- иначе если t равно ACC, это допуск;
- иначе если t равно POP, это выброс;
- иначе если t меньше либо равно MAX_RULE, это правило;
- иначе это ошибка управляющей таблицы.

Последнее возможно в случае ручного редактирования таблицы.

Теперь собственно действия.

В случае ошибки выводим сообщение (в поток parse_stream):

"**\nsyntaxan: failure synta = %d\n\n**"

(выводится переменная t) после чего завершаем метод, возвращаем 0.

В случае допуска выходим из цикла (break).

В случае выброса выталкиваем терминал из стека в at (at для контроля, что было вытолкнуто), после чего получаем очередной токен точно так, как это сделано в начале метода parse, с завершением и возвратом нуля в случае неудачи.

В случае выбора альтернативы (в случае правила) выталкиваем в переменную at нетерминал со стека (at опять только для контроля). Затем в переменную n получаем длину правила из массива RLEN. Переменная n для контроля. Затем формируем цикл, проталкивающий правило номер t в стек задом наперед. При этом в цикле для контроля в переменную bt записываем проталкиваемый символ правила, после чего проталкиваем его в стек. Нужно также сгенерировать последовательность правил. Есть два варианта: записывать правила в дополнительный стек, либо выводить их в какой-нибудь поток. Это точно понадобится, когда какой-то текст не будет допущен.

В случае ошибки синтаксической таблицы генерируем исключение:

```
throw "syntaxan: invalid SYNTA";
```

И это всё.

5.4. Лексический анализатор

Лексический анализатор должен распознавать ровно те типы лексем, что определяет SYNAX, не больше и не меньше. Любое отступление от этого правила приведет к неправильной работе транслятора.

Для Basic требуется, чтобы лексический анализатор распознавал перевод строки, токен LF, так как этот токен часть синтаксической грамматики.

Для Pascal недопустимы вещественные константы, начинающиеся с точки, для других языков допустимы. В этом, кстати, легко убедиться, если попробовать написать программу, например, на Pascal.

Нужно также уточнить список ключевых слов, разбираемых автоматом is_keyword. Разбираться должны ровно те ключевые слова, что есть в грамматике.

5.5. Проверка грамматики

Оставшаяся часть этой работы заключается в проверке грамматики. Для этого подаем на вход транслятора всевозможные тексты в соответствии с имеющимся воображением и убеждаемся, что они правильные или неправильные. Следует, однако, проверить все возможные конструкции, пытаясь найти ошибку. Это очень сложная и ответственная часть работы, нельзя к ней отнестись спустя рукава. Если ошибка обнаружится позднее, во время разработки генератора кода, придется переделывать все от начала до конца, и мало не покажется, это гарантируется.

5.6. Контрольные вопросы и упражнения

6. Работа ST-106. Анализатор LR(1)

Цели:
- построение анализатора грамматик класса LR(1).
Задачи:
- построение управляющей таблицы;
- разработка алгоритма анализатора LR.
Опорные документы:
[1]

6.1. Рабочее пространство

В модуле simplet.cpp нужно подключить модуль синтаксического анализатора LL(1), sa4LR.h, другие модули синтаксических анализаторов должны быть отключены (закомментированы). Модуль sa4LR.h включает в себя модуль управляющей таблицы syntalr.h.

6.2. Генерация файлов SYNAX

Открываем SYNAX и грамматику типа LR. Анализируем грамматику, переходим на вкладку Symbols, выполняем анализ и экспорт.

Копируем в файле экспорта константы перечисления sttype и вставляем в перечисление в модуле types.h. Следим за тем, чтобы перечисление начиналось следующим образом:

```
typedef enum sttype : stack_t {
    OUT_START = 997,
    TOK_COMMENT,
    TOK_UNKNOWN,
    // токены
    TOK_EOT = 0,
```

Никакие отклонения от порядка следования констант недопустимы.

Возможно, что придется некоторые константы токенов закомментировать или, наоборот, раскомментировать, если токены этой грамматики не совпадут с токенами предыдущей, что, вообще говоря, маловероятно.

В SYNAX переходим на вкладку SLR(1), анализируем и экспортируем. Находим файл экспорта и вставляем его целиком в модуль syntalr.h. Начало модуля при этом будем иметь примерно следующий вид:

```
// 2019 ReVoL Primer Template
// syntalr.h
// синтаксическая таблица LR

/* Generated by 2016 ReVoL SYNAX */
/* 00-00-2000 00:00:00 */
```

Далее расположена вспомогательная таблица длин правил:

```
/* Rule Length */
int RLEN[] = { 0,2,0,1,5,3,0,4,6,2,4, . . . };
```

Далее расположена вспомогательная таблица левых нетерминалов:

```
/* Rule Symbol */
sttype RSYM[] = {TOK_EOT,SYM_M,SYM_S,SYM_S,SYM_D_,SYM_D_, . . . };
```

Далее следуют три константы:

```
#define ACC 67 /* ACCEPT CODE  */
#define START SYM_M /* START SYMBOL */
#define MAX_DFA_STATE 66 /* LAST DFA STATE */
```

Первая задает код допуска, вторая определяет стартовый символ (аксиому грамматики), третья определяет номер последнего состояния ДКА, который разбирает активные префиксы на стеке МП-автомата.

Далее следует собственно управляющая таблица SYNTA, которая есть не что иное, как таблица переходов ДКА, в которую добавлена информация о свертках. Заметим, что эта таблица никогда не содержит синтаксических ошибок, но в конце файла экспорта все конфликты перечислены, если они есть, и на это нужно обратить внимание.

6.3. Лексический анализатор

При необходимости уточняем лексемы, распознаваемые лексическим анализатором. Обратим внимание также на ключевые слова, которые разбираются автоматом is_keyword.

6.4. Разработка алгоритма

Если проект компилируется, переходим к разработке модулю синтаксического анализатора SLR(1), sa4LR.h.

Требуемые структуры данных уровня класса:

- элемент данных tok типа sttoken. Это текущий токен потока.
- элемент данных sta типа ststack, параметризованный типом stack_t и константой MAX_STACK. Это рабочий стек МП-автомата.
- функция-элемент next_token:

```
// возвращает очередной токен
int next_token() {
    return lex->next_token(tok);
}
```

Метод parse. Он начинается с получения очередного (первого) токена при помощи метода next_token. Если next_token завершается неудачно, то функция завершается и возвращается 0.

Далее идут элементы данных, требуемые для работы МП-автомата:

- переменная m типа. Это новое состояние ДКА;
- переменная s типа stack_t. Это символ на стеке;
- переменная t типа stack_t. Это значение в таблице.

Далее проталкиваем на рабочий стек TOK_EOT (дно стека).

Далее следует бесконечный цикл автомата LR.

В цикле автомат выполняет следующие действия:

1) получить символ на вершине стека в s;

2) получить в переменную t значение в таблице, используя значение s как первый индекс, а значение tok.stt как второй индекс;

3) анализируя значение t, выполнить соответствующее действие. Это сделает оператор if, который мы рассмотрим далее.

После цикла в поток parse_stream выводится success и возвращается 1:

```
fprintf(parse_stream, "\nsyntaxan: success.\n\n");
return 1;
```

Остается разбор значения в таблице t. Описываем оператор if в точном соответствии с теорией (со значениями в управляющей таблице).

Заметим, что положительные значения в таблице соответствуют переносу, а отрицательные — свертке.

Условия оператора if:

- если t равно ACC, это допуск;
- иначе если t меньше нуля, это свертка;
- иначе если t больше нуля и меньше MAX_DFA_STATE, это перенос;
- иначе если t больше нуля, то это ошибка синтаксической таблицы;
- иначе это синтаксическая ошибка.

Теперь собственно действия.

В случае допуска выходим из цикла.

В случае свертки снимаем со стека RLEN[t] символов, получаем в s новое состояние на стеке, получаем в m новое значение в таблице, проталкиваем m в стек, выводим номер правила t.

В случае переноса проталкиваем на стек новое состояние t, получаем следующий токен при помощи next_token.

В случае ошибки синтаксической таблицы генерируем исключение:

```
throw "syntaxan: invalid SYNTA";
```

В случае синтаксической ошибки выводим в поток parse_stream сообщение, например:

"**\nsyntaxan: failure synta = %d\n\n**"

(выводится переменная t) после чего завершаем метод, возвращаем 0.

Это конец алгоритма, далее следует его тестирование.

6.5. Контрольные вопросы и упражнения

7. Работа ST-107. Трансляционная грамматика

Цели:
- изучение внутреннего представления программы в виде ПОЛИЗ.

Задачи:
- определение трансляций операторов;
- формирование ленты ПОЛИЗ.

Опорные документы:
[1]

7.1. Рабочее пространство

В модуле simplet.cpp нужно подключить модуль синтаксического анализатора LL(1), sa5LLT.h, другие модули синтаксических анализаторов должны быть отключены (закомментированы). Модуль sa5LLT.h включает в себя модуль управляющей таблицы syntallo.h.

Отлаженную функцию parse модуля sa4LL.h необходимо скопировать в модуль sa5LLT.h. Алгоритм разбора LL(1) грамматики останется практически прежним.

7.2. Представление программы в виде ПОЛИЗ

ПОЛИЗ (польская инверсная запись) — это такая форма внутреннего представления программы, в которой операция записывается в виде:

аргумент1, аргумент2, …, аргументN, операция

Иначе говоря, если у операции N аргументов (операция N-местная), то символу операции предшествуют N аргументов.

Будем называть всю последовательность операций, из которых состоит входная программа, лентой ПОЛИЗ, и цель данной работы — сформировать эту ленту. Сама по себе лента для простоты представляется как массив ограниченного размера, элемент данных класса ststrip.

Самым трудным сейчас будет понять, из каких операций складывается входная программа. Будем рассматривать по очереди все операторы, допустимые в программе. В качестве входного языка выберем Basic.

Первый оператор — объявление переменной:

Dim Id As Type

Заметим, что для простоты один оператор объявляет только одну переменную. Если объявляется несколько переменных одним оператором, то и лента формируется несколько иначе.

В чем заключается эта операция? Объявление переменной вводит в программу новый символ. Опять же для простоты, все символы программы считаются глобальными. Если это не так, то и структуры данных транслятора должны быть изменены, и каждому символу потребуется декорация.

Тогда действием является занесение идентификатора Id в таблицу символов, и приписывание к нему типа данных Type. Из этого следует, что в трансляторе должна быть таблица символов. Эта таблица есть, формирует ее класс stsyms, а элементами таблицы являются структуры типа stsym, в данном трансляторе заменяемые структурой токена sttoken.

Теперь рассмотрим, как должна выглядеть запись ПОЛИЗ.

Аргументами являются идентификатор Id и тип данных Type. Операцию удобно назвать DIM, тогда запись имеет вид:

Id, Type, DIM

Итак, оператор Dim транслируется в три элемента ленты ПОЛИЗ.

Каждый элемент ленты ПОЛИЗ — это структура данных, хранящая необходимую информацию. В данном трансляторе в качестве этой структуры выступает тип exstel (execute stack element), заменяемый опять же структурой токена sttoken.

Прежде всего нужно знать, что хранит элемент ленты. Для этого используется поле stt типа sttype структуры sttoken. Введем понятие операционного символа транслятора как некоторой константы, задающей тип элемента ПОЛИЗ. Эти константы включаем в перечисление типов sttype, и чтобы отличать их от других символов, TOK_XXX и SYM_XXX, будем называть их OUT_XXX. Все эти новые константы описываем за последним символом, сгенерированным генератором SYNAX.

Рассмотрим первый элемент записи, идентификатор. В момент разбора оператора объявления идентификатор хранит текущий токен трансляции, то есть переменная tok. Получается, что удобно просто взять этот токен и записать его на ленту как есть, и его тип окажется TOK_ID.

Рассмотрим второй элемент. Это тип данных. В грамматике может быть написано что угодно, но в тексте программы будет написано вполне конкретно, например:

Dim A As Long

Таким образом, второй элемент ленты — опять же текущий токен трансляции, который точно также можно записать на ленту, и его тип будет вполне конкретным, TOK_LONG.

Остается записать на ленту элемент самой операции, для которой удобно ввести новую константу OUT_DIM.

Таким образом, на ленте ПОЛИЗ должны быть сформированы элементы TOK_ID, TOK_LONG и OUT_DIM.

Теперь поговорим о том, как элементы записи появятся на ленте. Для этого и нужно понятие операционного символа, который вставляется в правило в нужном месте (и определение этого места самая сложная задача). Символ операционный, потому что он заставляет анализатор LL(1) грамматики выполнить дополнительное действие, операцию, а именно, вывод на ленту определенного элемента.

Рассмотрим правило, описывающее оператор объявления.

Пусть в грамматике написано:

S→ dim id as T lf

T→ integer

T→ long

Здесь нетерминал T обозначает тип данных.

Добавление анализатором текущего токена id на ленту опишем, как операционный символ OUT_ID. Тогда первое правило грамматики должно содержать этот символ непосредственно за токеном id:

S→ dim id OUT_ID as T lf

В тот момент, когда анализатор обнаружит идентификатор на входе, он выталкивает из стека символ TOK_ID, при этом текущий токен еще актуален, но дальше анализатор получает следующий токен, и токен идентификатора исчезает. В промежуток между выталкиванием символа со стека и получением нового токена анализатор должен проверить, есть ли на стеке операционные символы, их все равно нужно удалять, иначе они нарушат работу анализатора. На стеке же обнаружится операционный символ OUT_ID, который запишет текущий токен на ленту, после чего токен будет заменен следующим в потоке. Таков механизм действия операционного символа.

Теперь в правила для нетерминала T нужно вписать операционные символы:

T→ integer OUT_INTEGER

T→ long OUT_LONG

Когда анализатор разберет токен, например, long, он точно также, как и в предыдущем случае, выведет на ленту текущий токен. Можно, кстати, вместо этих операционных символов использовать один OUT_TYPE, технически это все равно, но ленту читать будет сложнее.

Наконец, в первое правило вставляем символ OUT_DIM, который выведет на ленту токен с типом OUT_DIM:

S→ dim id OUT_ID as T OUT_DIM lf

Таким образом, наша задача заключается в том, чтобы вставить в правила операционные символы, и модернизировать анализатор так, чтобы он по крайней мере выбрасывал их со стека.

Примерно таким же образом разбираются все другие операторы, которые могут встретиться во входном языке, и правила для них корректируются операционными символами. И у нас появляется огромная проблема с разработкой транслятора. Стоит нам хоть немного изменить грамматику, генератор SYNAX сгенерирует новую таблицу правил, в которой операционных символов не будет и их нужно будет вписывать заново.

Вероятно, существуют генераторы компиляторов, которые позволяют записывать операционные символы непосредственно в грамматику. Такая грамматика, кстати, называется трансляционной. Имеющаяся версия SYNAX трансляционные грамматики не поддерживает.

7.3. Входной язык трансляции

Поскольку стоящая перед нами задачи очень сложна, все используют входной язык, заданный следующей грамматикой:

```
#0 <M>
#1 <M>=<S><M>
#2 <M>=.
#3 <S>=[LF]
#4 <S>=[dim][id][as][long][LF]
#5 <S>=[id][=]<E>[LF]
#6 <S>=[if]<E>[then][LF]<M><A>
#7 <A>=[else][LF]<M>[end][if][LF]
#8 <A>=[end][if][LF]
#9 <S>=[print]<E>[LF]
#10 <E>=<R><R.>
#11 <R.>=[=]<R><R.>
#12 <R.>=[<>]<R><R.>
#13 <R.>=[<]<R><R.>
#14 <R.>=[>]<R><R.>
#15 <R.>=.
#16 <R>=<T><T.>
#17 <T.>=[+]<T><T.>
#18 <T.>=[-]<T><T.>
#19 <T.>=.
#20 <T>=<P><P.>
#21 <P.>=[*]<P><P.>
#22 <P.>=[/]<P><P.>
#23 <P.>=.
#24 <P>=[id]
#25 <P>=[I4]
#26 <P>=[(]<E>[)]
```

Эту грамматику нужно вписать в SYNAX, сохранить в файл с именем basicone. Далее нужно выполнить анализ грамматики, анализ LL(1), экспорт символов и экспорт управляющей таблицы. Экспорт символов вставляем в перечисление sttype, а таблицу — в модуль syntallo.h.

7.4. Лексический анализатор

Теперь нужно настроить лексический анализатор на распознавание только тех лексем, что есть в языке. Прежде нужно убедиться, что все идентификаторы приводятся к нижнему регистру.

Перечень ключевых слов: dim, as, long, if, then, else, end, print.

Перечень операций и пунктуаторов: +, –, *, /, (,), =, >, <>, <.

Заметим, что операция присваивания и равенства задаются одним и тем же символом (знаком равно) и константой TOK_EQ.

Для тестирования можно использовать следующий входной файл:

```
Dim A As Long
A = 1
If A <> 0 Then
    Print 3 + A * 5
Else
    Print 0
End If
```

7.5. Исключение операционных символов

Нам нужен механизм выявления операционных символов на стеке.

Модуль sa5LLT.h, класс syntaxan.

Сначала нам нужен метод, который примет выводимый символ:

```
private:
    // запись символа на ленту ПОЛИЗ
    void out(sttype tt) {
        int i = 0, j = 0;
        sttoken t = tok;
    }
```

Затем описываем метод clear_stack:

```
    // выводит операционные символы со стека
    stack_t clear_stack() {
        stack_t s;
        while ((s = sta.top()) > SYM_LAST {
            s = sta.pop();
            out((sttype)s);
        }
        return s;
    }
```

Теперь метод clear_stack нужно вызывать в двух местах.

1. В самом начале рабочего цикла:

```
        // рабочий цикл автомата LL
        while (1) {
            // состояние на стеке
            s = clear_stack();
            // значение в таблице
            t = SYNTA[s][tok.stt];
```

2. Непосредственно после выталкивания со стека терминала:

```
            } else if (t == POP) {
                // ВЫБРОС
                // выталкиваем терминал из рабочего стека
                at = sta.pop();
                // выталкиваем операционные символы
                clear_stack();
                // следующий токен
                if (!next_token()) return 0;
```

После этой модификации разбор должен происходить успешно при любых дополнительных символах в правилах.

7.6. Формирование ПОЛИЗ оператора объявления

В конец перечисления sttype добавляем операционные символы:

```
SYM_LAST = SYM_P_,
// конец символов грамматики
// символы генератора кода

OUT_END,
OUT_ID,
OUT_LONG,
OUT_DIM,
```

Вставляем операционные символы в правило 4 грамматики:

```
/* 4 SYM_S  */{ TOK_DIM, TOK_ID, OUT_ID, TOK_AS, TOK_LONG, OUT_LONG,
OUT_DIM, TOK_LF, TOK_EOT },
```

Убеждаемся, что символы выводятся из стека (разбор успешен).

7.7. Лента ПОЛИЗ

Нам нужна собственно лента ПОЛИЗ.

Объявляем в классе syntaxan следующие структуры данных:

```
// текущий токен
sttoken tok;
// таблица символов
stsyms<MAX_SYMS> syms;
// исполняющий стек
exstack<MAX_EXEST> exe;
// лента ПОЛИЗ
ststrip<MAX_STRIP> strip;
// стек меток
ststack<int, MAX_LABEL> stl;
// стек МП-автомата
ststack<stack_t, MAX_STACK> sta;
```

Собственно лента — это переменная strip.

Возвращаемся к МП-автомату. Когда работа автомата успешно завершается, запишем на ленту символ конца ленты и выведем содержимое ленты в поток parse_srteam:

```
        } // while (1)
        // записываем конец ленты ПОЛИЗ
        out(OUT_END);
        // выводим содержимое ленты ПОЛИЗ
        strip.print(parse_stream);
        fprintf(parse_stream, "\nsyntaxan: success.\n\n");
        return 1;
    } // parse()
```

7.8. Метод out

На вход метода поступает тип символа tt. На данном этапе достаточно поменять значение stt и добавить символ на ленту:

```
    void out(sttype tt) {
        int i = 0, j = 0;
        sttoken t = tok;
        t.stt = tt;
        strip.add(t);
    }
```

Запускаем программу, убеждаемся, что лента ПОЛИЗ формируется в соответствии с нашим представлением:

```
001 ID a
002 LONG
003 DIM
004 END
```

7.9. Формирование ПОЛИЗ оператора присваивания

ПОЛИЗ оператора присваивания для нашей грамматики имеет вид
ID выражение OUT_ASS
Формируем правило для оператора присваивания:

```
/* 5 SYM_S  */{ TOK_ID, OUT_ID, TOK_EQ, SYM_E, OUT_ASS, TOK_LF,
TOK_EOT },
```

В перечисление sttype добавляем символ OUT_ASS. Убеждаемся, что формируется ПОЛИЗ:

```
001 ID a
002 LONG
003 DIM
004 ID a
005 ASS
006 END
```

Вычисление ПОЛИЗ выражения составляет основную часть оператора присваивания. Если, например, выражение задано как 2+3*4, то должна быть сформирована последовательность элементов ПОЛИЗ 2 3 4 * +.

Добавляем в перечисление sttype символ OUT_I4. Элементы выражения формируются операционными символами OUT_ID или OUT_I4 в правилах для нетерминала P:

```
/*24 SYM_P  */{ TOK_ID, OUT_ID, TOK_EOT },
/*25 SYM_P  */{ TOK_I4, OUT_I4, TOK_EOT },
```

Для выполнения каждой из операций в перечисление sttype добавляем символы OUT_EQ, OUT_NE, OUT_LT, OUT_GT, OUT_ADD, OUT SUB, OUT_MUL и OUT_DIV.

Записываем все восемь операций.

Пример для операции равенства:

```
/*11 SYM_R_ */{ TOK_EQ, SYM_R, SYM_R_, OUT_EQ, TOK_EOT },
```

Другие операции записываются аналогично.

Добавляем в перечисление sttype символ OUT_PRINT.

В правило 9 записываем этот символ:

```
/* 9 SYM_S  */{ TOK_PRINT, SYM_E, OUT_PRINT, TOK_LF, TOK_EOT },
```

В результате всех изменений должна формироваться следующая лента ПОЛИЗ:

```
001 ID a
002 LONG
003 DIM
004 ID a
005 I4 1
006 ASS
007 ID a
008 I4 0
009 NE
010 I4 3
011 ID a
012 I4 5
013 MUL
014 ADD
015 PRINT
016 I4 0
017 PRINT
018 END
```

Очевидно, что все операции сформированы, но нет управления потоком их выполнения.

7.10. ПОЛИЗ условного оператора

Условный оператор транслируется гораздо сложнее, так как он требует линеаризации веток. Операция NE сравнивает идентификатор a со значением 0 и выполняет одну из веток: 010-015 в случае истинного значения оператора NE, или 016-017 в случае ложного значения. Для этого в поток элементов ПОЛИЗ нужно вставить метки и операции перехода BZ и BR.

Общий вид ПОЛИЗ условного оператора следующий:

E L1 BZ ветка-истина L2 BR L1 DEFL ветка-ложь L2 DEFL

Сначала вычисляется выражение E, результат которого либо ложный, либо истинный. Затем генерируется метка L1 и операция условного перехода по лжи BZ. Далее на ленте следуют операции ветки, соответствующей истинности операции NE, заканчивающиеся генерацией метки L2 и операции безусловного перехода BR. Далее определяется метка L1 при помощи операционного символа OUT_DEFL. Далее следует ветка лжи и определение метки L2.

Описываем в перечислении sttype символы OUT_LABEL, OUT_BZ, OUT_BR, OUT_PUSH, OUT_POPL, OUT_SWAP, OUT_DEFL.

Метод out должен выполнять определенные действия для операций OUT_PUSH, OUT_POPL и OUT_SWAP. В случае OUT_PUSH нужно сгенерировать новую метку, протолкнуть ее в стек меток и записать на ленту:

```
void out(sttype tt) {
    int i = 0, j = 0;
    sttoken t = tok;
    t.stt = tt;
    switch (tt) {
    case OUT_PUSP:
        j = strip.new_label();
        stl.push(j);
        t.int_val = j;
        t.stt = OUT_LABEL;
        break;
    }
    strip.add(t);
}
```

В случае OUT_POPL нужно вытолкнуть метку со стека меток и записать ее на ленту:

```
    case OUT_POPL:
        j = stl.pop();
        t.int_val = j;
        t.stt = OUT_LABEL;
        break;
```

В случае OUT_SWAP нужно две верхние метки стека меток поменять местами, вытолкнуть верхнюю метку и записать ее на ленту:

```
    case OUT_SWAP:
        i = stl.pop();
        j = stl.pop();
        stl.push(i);
        t.int_val = j;
        t.stt = OUT_LABEL;
        break;
```

Теперь можно приступить к добавлению символов в правила.

Правило 6:

```
/* 6 SYM_S  */{ TOK_IF, SYM_E, TOK_THEN, TOK_LF, OUT_PUSH, OUT_BZ,
SYM_M, SYM_A, TOK_EOT },
```

Правило 7:

```
/* 7 SYM_A  */{ TOK_ELSE, TOK_LF, OUT_PUSH, OUT_BR, SYM_M, TOK_END,
TOK_IF, TOK_LF, TOK_EOT },
```

Мы вставили переходы перед ветками истины и лжи. Запуск программы должен убедить нас, что перед этими ветками вставились метки с номерами 1 и 2 и соответствующие операции переходов.

Правило 7:

```
/* 7 SYM_A  */{ TOK_ELSE, TOK_LF, OUT_PUSH, OUT_BR, OUT_SWAP,
OUT_DEFL, SYM_M, TOK_END, TOK_IF, TOK_LF, TOK_EOT },
```

Мы вставили определение метки 1, сгенерированной в правиле 6.

В результате должна получиться следующая последовательность:

```
018 LABEL 2
019 BR
020 LABEL 1
021 DEFL
022 I4 0
023 PRINT
024 END
```

Таким образом, ветки лжи получила метку 1.

Далее нужно сгенерировать метку окончания условного оператора. Она будет разной для разных случаев, когда else есть, и когда else нет.

Правило 7 (есть else):

```
/* 7 SYM_A  */{ TOK_ELSE, TOK_LF, OUT_PUSH, OUT_BR, OUT_SWAP,
OUT_DEFL, SYM_M, OUT_POPL, OUT_DEFL, TOK_END, TOK_IF, TOK_LF,
TOK_EOT },
```

Правило 8 (нет else):

```
/* 8 SYM_A  */{ OUT_POPL, OUT_DEFL, TOK_END, TOK_IF, TOK_LF, TOK_EOT
},
```

В результате должна быть сгенерирована лента:

```
001 ID a
002 LONG
003 DIM
004 ID a
005 I4 1
006 ASS
007 ID a
008 I4 0
009 NE
010 LABEL 1
011 BZ
012 I4 3
013 ID a
014 I4 5
015 MUL
016 ADD
017 PRINT
018 LABEL 2
019 BR
020 LABEL 1
021 DEFL
022 I4 0
023 PRINT
024 LABEL 2
025 DEFL
026 END
```

Заметим, что операционные символы могут быть вставлены и другим образом, так, что обе метки будут генерироваться в любом случае.

8. Работа ST-108. Интерпретация ленты ПОЛИЗ

Цели:
- интерпретация ленты ПОЛИЗ.
Задачи:
- чтение ленты ПОЛИЗ и выполнение команд;
- семантический анализ.
Опорные документы:
[1]

8.1. Рабочее пространство

Эта работа является продолжением предыдущей.

Для выполнения команд ленты ПОЛИЗ добавим в класс syntaxan метод execute:

```
// исполняющая машина ПОЛИЗ
void execute() {
}
```

Этот метод вызовем в завершении метода parse:

```
    strip.print(parse_stream);
    execute();
    fprintf(parse_stream, "\nsyntaxan: success.\n\n");
    return 1;
} // parse
```

Для выполнения команд ПОЛИЗ требуется вычислительный стек:

```
// таблица символов
stsyms<MAX_SYMS> syms;
// исполняющий стек
exstack<MAX_EXEST> exe;
// лента ПОЛИЗ
ststrip<MAX_STRIP> strip;
// стек меток
ststack<int, MAX_LABEL> stl;
// стек МП-автомата
ststack<stack_t, MAX_STACK> sta;
```

Элементами вычислительного стека являются структуры типа exstel, эквивалентного в данном трансляторе типу sttoken.

8.2. Семантический анализ

В данном трансляторе выполняется семантический анализ, связанный с объявлением переменных. Проверяются следующие два семантических правила:
- наличие объявления переменной;
- отсутствие повторного объявления одной и той же переменной.

Семантические правила проверяются при выполнении ленты ПОЛИЗ по мере возникновения необходимости. Результатом невыполнения любого из правил будет выбрасывание соответствующего исключения.

8.3. Принципы выполнения команд ПОЛИЗ

На ленте ПОЛИЗ есть операнды и команды. Читаем ленту, начиная с начала. Если на ленте операнд, проталкиваем его в стек. Если на ленте команда, она имеет нуль или более операндов. Если операндов нет, выполняем команду. Если операнды есть, извлекаем их из стека в локальные переменные, выполняем операцию, результат помещаем на стек.

Формируем структуру метода execute. Нам требуются:

- указатель текущего положения на ленте strip_pointer типа int;
- тип текущего элемента ленты stt типа sttype;
- два элемента X и Y типа exstel для выполнения операций;
- вспомогательная переменная j типа int;
- счетчик операций it_counter типа int:

```
// счетчик операций ПОЛИЗ
int it_counter = 1;
// указатель ленты ПОЛИЗ
int strip_pointer = 1;
// текущий элемент ПОЛИЗ
sttype stt;
// переменные для вычислений
evstel X, Y;
// вспомогательные
int j = 0;
```

Далее нужно сформировать цикл чтения элемента ленты и выполнение связанных с ним действий. Цикл включает в себя проверку максимального числа операций для исключения ее зацикливания, заданное константой MAX_IT. Описываем цикл чтения элементов ленты:

```
// рабочий цикл выполнения ленты ПОЛИЗ
while (1) {
    // очистим переменные
    X.reset();
    Y.reset();
    // считываем тип элемента ленты
    stt = strip[strip_pointer].stt;
    switch (stt) {
    case OUT_END:
        fprintf(error_stream, "EXE DONE\n\n");
        return;
    }
    if (++it_counter > MAX_IT) {
        throw "exe deadlock";
    }
```

Далее описываем возможные элементы ленты в операторе switch.

8.4. Вспомогательный метод вычислительного стека

Нам потребуется вспомогательный метод, который извлекает из вычислительного стека значение. Это необходимо потому, что на стеке могут находиться элементы OUT_I4, являющиеся значениями, или OUT_ID, являющиеся именами переменных. Весь смысл метода заключается в получении значения, соответствующего элементу OUT_ID:

```
// извлекает из стека значение
void exe_pop(exstel & e) {
    exe.pop(e);
    if (e.stt == OUT_I4) {
    } else if (e.stt == OUT_ID) {
        int j = syms.find(e);
        if (j == ST_NOTFOUND) {
            // символ не найден
            throw "exe_pop identifier not found";
        }
        // извлекаем значение из таблицы символов
        e.int_val = syms[j].int_val;
        // меняем тип элемента на константный
        e.stt = OUT_I4;
    } else {
        // неправильная лента ПОЛИЗ
        throw "exe_pop internal error";
    }
}
```

8.5. Оператор объявления переменной

Операционному символу OUT_DIM на ленте предшествуют символы OUT_ID и OUT_LONG. Кроме того, на ленте встречаются также константы OUT_I4. Все эти элементы являются операндами, поэтому для них описываем проталкивание в стек следующим образом:

```
case OUT_ID: case OUT_I4: case OUT_LONG:
    // проталкиваем в стек
    exe.push(strip[strip_pointer]);
    // следующий элемент ленты
    strip_pointer++;
    break;
```

Для элемента OUT_DIM выполняем следующие действия:
- выталкиваем из стека элемент в Y (методом exe.pop);
- выталкиваем из стека элемент в X (методом exe.pop);
- добавляем элемент X в таблицу символов методом syms.insert, результат принимаем в переменную j.
- если j равно ST_EXISTS, выбрасываем исключение со строкой: "exe duplicate declaration".
- устанавливаем тип данных Y для переменной X (через j);
- перемещаем указатель ленты strip_pointer.

Мы не устанавливаем тип данных Y, так как у нас всего один тип данных в трансляторе (пропускаем это действие).

Отлаживаем эту часть, отслеживая пошаговым выполнением, как в вычислительный стек заносятся операнды, а затем как в таблицу символов заносятся переменные.

8.6. Оператор присваивания

Элементу OUT_ASS предшествуют элемент OUT_ID и элемент, являющийся результатом вычисления выражения. Это может быть как элемент OUT_ID, так и элемент OUT_I4, причем последний может быть задан непосредственно в операторе, например, a = 1, или может быть результатом вычислений, например, a = 1 + 1.

Поэтому:

- выталкиваем из стека в Y значение методом exe_pop;
- выталкиваем из стека в X переменную методом exe.pop;
- ищем переменную X в таблице символов методом syms.find, результат принимаем в переменную j;
- если j равно ST_NOTFOUND, выбрасываем исключение со строкой: "exe identifier not found";
- присваиваем переменной X значение Y:

```
// присвоить переменной значение
syms[j].int_val = Y.int_val;
```

- перемещаем указатель ленты strip_pointer.

8.7. Операции

Вот где начинается главное веселье.

У нас в языке 8 операций, и каждую нужно описать. Что с этим связано? В нашем простом трансляторе практически ничего. Но на практике поднимаются следующие вопросы.

1. Являются ли значения совместимыми? Например, можно ли сложить целое число 1 со строкой "a", если да, то каков результат?
2. Если два операнда имеют разные, но совместимые типы, то каков должен быть тип результата, требуется ли приведение типов к одному?
3. Можно ли складывать целочисленные и логические значения?
4. Является ли логический тип отдельным типом, или его можно трактовать как арифметическое значение?

Все эти вопросы не могут быть решены синтаксической грамматикой, поэтому это семантический анализ.

В нашем трансляторе один тип данных, и это счастье, потому что у нас не возникает указанных вопросов, также еще и потому, что логическое значение можно трактовать как арифметическое и складывать его с целочисленным. Поэтому предельно просто.

Все операции сначала объединяем в один case:

```
case OUT_ADD: case OUT_SUB: case OUT_MUL: case OUT_DIV:
case OUT_EQ:  case OUT_NE:  case OUT_LT:  case OUT_GT:
```

Общим для этих операций является то, что все они двухместные.

Поэтому общие действия операций следующие:

- выталкиваем в Y значение правого операнда методом exe_pop;
- выталкиваем в X значение левого операнда методом exe_pop;

Далее операции нужно разъединить внутренним оператором switch, и описать выполнение каждой операции отдельно, стараясь не перепутать левый и правый операнды.

Вот, например, как выполняется операция сложения OUT_ADD:

```
// выполнить операцию
switch (stt) {
case OUT_ADD:
    X.int_val += Y.int_val;
    break;
```

При выполнении операции деления необходимо проверить значение делителя, и если он равен нулю, выбросить исключение со строкой:

"exe division by zero"

После того, как каждая из операций будет выполнена, вновь соединяем операции и выполняем следующие действия:

- проталкиваем на стек элемент X (результат операции);
- перемещаем указатель ленты strip_pointer.

Здесь нужно тестировать, исполняя программу шаг за шагом.

8.8. Операции, связанные с метками

Как ни странно, но на ленте нет никаких if, while и прочей ветвящей петрушки, потому что все это выдумки создателей языков, никакая исполняющая машина не имеет этих операторов, потому что исполняющая машина — это операции и переходы (что с точки зрения машины тоже операции). И нам предстоит разобраться с переходами и их метками.

Если на ленте находится метка OUT_LABEL, то нужно сформировать целочисленное значение метки в операнде X, после чего протолкнуть X на стек методом exe.push, переместить указатель ленты:

```
casa OUT_LABEL:
    // на стек помещаем значение метки
    X.int_val = strip[strip_pointer].int_val;
    // тип значения - константа
    X.stt = OUT_I4;
    exe.push(X);
    strip_pointer++;
    break;
```

Если на ленте находится команда OUT_DEFL, то нужно вытолкнуть в X методом exe.pop метку, помещенную на стек предыдущим действием, так как при обычном чтении ленты определение метки никаких действий не предполагает. После этого нужно переместить указатель ленты.

Если на ленте условный переход OUT_BZ, то:

- выталкиваем в Y значение метки методом exe_pop;
- выталкиваем в X значение выражения методом exe_pop;
- если целочисленное значение X равно нулю (ложь), то:
    - ищем на ленте метку методом strip.find_DEF(Y.int_val), результат принимаем в переменную j.
    - если j равно –1, то выбрасываем исключение со строкой "exe label not found"
    - устанавливаем значение strip_pointer равным j.
- иначе:
    - перемещаем указатель strip_pointer обычным образом.

Если на ленте безусловный переход OUT_BR, то:

- выталкиваем в Y значение метки методом exe_pop;
- ищем на ленте метку методом strip.find_DEF(Y.int_val), результат принимаем в переменную j;
- если j равно –1, то выбрасываем исключение со строкой "exe label not found";
- устанавливаем значение strip_pointer равным j.

8.9. Оператор печати

Остается запрограммировать команду печати OUT_PRINT.

Здесь все просто. Выталкиваем значение в Y, выводим результат в поток parse_stream при помощи функции fprintf, используя управляющую строку "exe print %d\n". После этого нужно перевести указатель ленты.